

Regularizing Recurrent Networks—On Injected Noise and Norm-based Methods

Saahil Ognawala

Technische Universität München

OGNAWALA@IN.TUM.DE

Justin Bayer

Technische Universität München

BAYERJ@IN.TUM.DE

Abstract

Advancements in parallel processing have lead to a surge in multilayer perceptrons' (MLP) applications and deep learning in the past decades. Recurrent Neural Networks (RNNs) give additional representational power to feedforward MLPs by providing a way to treat sequential data. However, RNNs are hard to train using conventional error backpropagation methods because of the difficulty in relating inputs over many time-steps. Regularization approaches from MLP sphere, like dropout and noisy weight training, have been insufficiently applied and tested on simple RNNs. Moreover, solutions have been proposed to improve convergence in RNNs but not enough to improve the long term dependency remembering capabilities thereof.

In this study, we aim to empirically evaluate the remembering and generalization ability of RNNs on polyphonic musical datasets. The models are trained with injected noise, random dropout, norm-based regularizers and their respective performances compared to well-initialized plain RNNs and advanced regularization methods like fast-dropout. We conclude with evidence that training with noise does not improve performance as conjectured by a few works in RNN optimization before ours.

1. Introduction

Recurrent Neural Networks are variations of multilayer perceptrons based function approximators, which are used to predict on time-series data. Such data may be text information in various languages, a musical sequence, a video, or a trend analysis in the financial

domain. As training for MLP goes, the most popular techniques are all based on some form of backpropagation of weight gradients (Rumelhart et al. (1988)). To train an RNN, the backpropagation of gradients is performed in time, on a time-unfolded representation of the network.

When such a time series network is trained by traditional backpropagation on error gradients, it suffers from one of two peculiar analytical problems—exploding gradients or vanishing gradients. When the error gradients are backpropagated through what is essentially a set of identical weight vectors, the gradients may grow smaller (vanishing gradients) or larger (exploding gradients) exponentially fast, until they become insignificant for training purpose or lead to instability. Conceptually, the problem of vanishing gradients exists in any deep neural network that relies on propagating its error downwards to train the weights. This issue is particularly harmful in case of RNN because it damages the capability of a network to learn properties of the problem that are *long-term dependent*. In simple terms, this means that due to its inherent nature of being time-series, a recurrent network needs to store not only the state representation of the input at time, t , but also of those seen at $t' < t$. This problem, in presence of vanishing gradients, becomes intractable for $t - t'$ exceeding a few dozens.

Due to the unstable behaviour of RNNs in dynamic space, they were not touched upon extensively until some sophisticated second-order optimization methods were introduced for feedforward neural networks (Martens, 2010), that were extended to RNNs. Also groundbreaking have been the advances in form of structural solutions like Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) that established state-of-the-art results on text prediction tasks, pathological tasks and such.

Till date, there have been no empirical studies on

claims as the ones made in Pascanu et al. (2012a) that regularization of recurrent weights by means of restricting the growth of $\frac{\partial x_t}{\partial x_{t'}}$ will fail to prevent vanishing gradients. There have also not been evaluations on the standard regularization-for-overfitting techniques in MLP training applied to RNN for remembering long term dependencies. In this study, we aim to evaluate the effect of norm-based regularization methods, artificial noise injection and dropout in weights before propagating derivatives on the ability of the network to remember long term dependencies as well as convergence.

2. Related Work

Bengio et al. (2013) present an experimental study that discusses the latest optimization trends in RNNs, including gradient clipping, second order optimization methods like Hessian-free, leaky integration units (LSTMs are also discussed as a part of this), momentum tricks in simple gradient descent (SGD), powerful output probability models based on deterministic variations of Restricted Boltzmann Machines and using sparse gradients as a regularization trick. The evaluations presented in the paper above are on the same music datasets that we use in our study, in addition to the Penn Treebank Corpus of text data.

Maas et al. (2012) describe deep recurrent networks that consist of denoising autoencoders (Vincent et al., 2008) at each time-step, to extract rich features out of audio signals by learning time-series representations from deliberately noise-ed input. The noise itself is not modelled by the autoencoder, which is the key idea behind learning a denoised input representation.

RNNs are typically described as a set of three transition functions, viz. input-to-hidden, hidden-to-hidden and hidden-to-output. Pascanu et al. (2013) delve into the matter of “depth” in RNNs by describing and evaluating the workings of an RNN when one or more of these three transitions are made deeper than a single layer.

The study by Hochreiter & Schmidhuber (1997) is a solution to the long term dependency problem in RNNs. In this, the authors propose a structural variation of a conventional RNN where, by adding additional short-term memory units that fire randomly, the long time-delay remembering capability of an RNN increases significantly. Graves (2013) extended the study of LSTMs by applying the idea to generate complex sequences of words in a text corpus, and handwriting patterns learned from real-valued positional information in calligraphy. Zaremba et al. (2014) improved

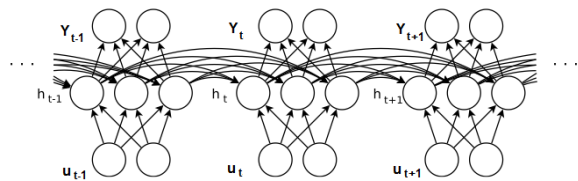


Figure 1. RNN unfolded in time. (Adapted from Sutskever (2013) with permission.)

generalization in LSTMs by applying Dropout (Hinton et al., 2012) only to the non-recurrent connections.

Murray & Edwards (1994) present an analysis of noisy MLP training models, where the cost function is appended with a noise term to improve trajectory of the training curve, generalization of the network and increase fault tolerance from data. The results were shown to be particularly useful in the field of VLSI network design.

The study of Jim et al. (1996) attempts to extend the noisy gradient descent model from feedforward networks to RNNs. The authors focus on convergence of RNNs, rather than the long term dependency problem. The noisy update model is applied to automata solving problems, which typically do not have pathologically long sequences that need to be remembered at arbitrary time delays.

In an analysis by Schaefer et al. (2008), the authors claim that the widely discussed problem of long-term dependency identification in RNNs does not really exist. This claim is validated by working a pathological sequence task through an RNN, and demonstrating its performance on increasing time delays between the relevant input and output values. However, this study does not present results on standard audio, video or text corpus data that are used in other pertinent publications in RNN.

3. Formulation of RNNs

RNNs are semantically applicable to tasks that are based on temporal consistency. Other than universal function approximators, a way of looking at MLPs is as orthogonal representation of the input features. RNNs exploit this representation technique by duplicating hidden layers of MLP in time-steps and fully connecting the consecutive hidden layers in time. Therefore, we get an unfolded representation of RNNs in time as shown in Fig. 1.

We, hence, define an RNN as

$$Y_t = \sigma_o(W_{ho}X_t + b_o) \quad (1)$$

$$X_t = \sigma(W_{hh}X_{t-1} + W_{ih}u_t + b_h) \quad (2)$$

At time-step t , u_t is the input, X_t is the activation of the hidden layer, h_t , and Y_t is the output of the network. The complete parameter set of the model is given by the input-to-hidden weights, W_{ih} , hidden-to-hidden weights, W_{hh} , hidden-to-output weights, W_{ho} , hidden layer bias, b_h , and output layer bias, b_o . σ_o and σ are the non-linear activation functions at the output and hidden layers, respectively.

4. Exploding Gradients and Effect on Long-term Dependencies

Bengio et al. (1994) and Pascanu et al. (2012a) explain the dynamics of the weight training using backpropagation through time in RNNs.

Consider the error function, ε , applied on the outputs of RNN. Calculating error gradient

$$\frac{\partial \varepsilon}{\partial \theta} = \sum_{1 \leq t \leq T} \frac{\partial \varepsilon_t}{\partial \theta} \quad (3)$$

$$\frac{\partial \varepsilon_t}{\partial \theta} = \sum_{1 \leq k \leq t} \frac{\partial \varepsilon_t}{\partial X_t} \frac{\partial X_t}{\partial X_k} \frac{\partial X_k}{\partial \theta} \quad (4)$$

Where, θ is the concatenated matrix of W_{ih} , b_h , W_{hh} , W_{ho} and b_o .

It is clear from Eq. (4) that derivative of loss function at every time-step, t , is affected by the activations at time-steps $k < t$.

Furthermore, consider the term $(\partial X_t / \partial X_k)$ on the right hand side of Eq. (4)

$$\frac{\partial X_t}{\partial X_k} = \prod_{t \geq j \geq k+1} \frac{\partial X_j}{\partial X_{j-1}} \quad (5)$$

The multiplication of the real valued derivatives at time-steps $k < t$ successively for all indices t in Eq. (4) may lead to the norm of the product growing very large or vanishing to zero, exponentially fast in time. This is harmful as far as storing long term time dependencies goes, because by the time the error gradient at k would have been propagated to $j \ll k$, the norm explosion or vanishing may have made the training regime unsuitable for any meaningful updates.

This compounding of the error gradient can happen in one of two opposite directions, both depending on the largest eigenvalue (spectral-radius), ρ , of the recurrent weight matrix. If the spectral radius is much

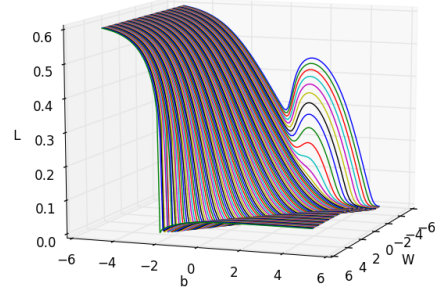


Figure 2. Training regime of a simple RNN. W : weight, b : bias, L : squared loss

less than 1, the gradient might vanish over time (if using a sigmoid-like non-linearity). On the other hand, if the spectral radius is bigger than 1, the gradient might explode over time.

5. Demonstration with a Simple Regime

Let us demonstrate the delicate nature of training a recurrent weight matrix, using an over-simplified architecture (a more expansive explanation, also from a dynamical systems perspective, can be found in Pascanu et al. (2012b)).

In Eq. 2, assume that there is no new input coming at every time-step, so that the second term with u_t becomes unnecessary. Furthermore, assume that X is a single dimension variable, which means that W_{hh} and b_h have dimensions $[1, 1]$ and $[1]$ respectively.

Our objective, then, is to start x with a zero value and reach a given target value, z , in a set number of time-steps. Fig. 2 shows the training graph over 10000 different initialization sets of W and b . On the third axis, L represents the squared loss of the model.

The steep wall perpendicular to the parameter space represents an explosion in gradients of the loss function. When the largest eigenvalue in the parameter matrix explodes, the curvature of the error surface compounds too, which is what the wall illustrates.

The thing most noteworthy is that when the search routine is at a point on the top surface of the error curve, it makes its next step in a direction perpendicular to the face of the wall. Depending on the learning rate, it might then fall to ground beyond the valley

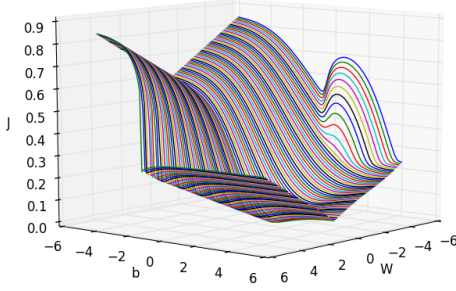


Figure 3. More desirable training regime of a simple RNN.

where the error reaches its minimum. This is not such a big problem, because the search must come back to the valley region, given itself to explore the ground region. Note, however, that is only until the search direction collides onto the wall again, at which point a small change in the norm of the update would take the search back to the top of the hill to repeat the entire search process.

The key, then, is to have a method that would smoothen the minima valley and decrease the slope of the steep wall so as to allow optimization to move in a less arbitrary fashion given a sufficiently small learning rate. A more acceptable routine may look like the one shown in Fig. 3.

6. Existing Solutions

6.1. Initialization and Momentum Tricks

Momentum (Polyak (1964)) with SGD method has the added advantage of preserving the directions of consistent change over multiple updates. The persistent change in directions can be thought of as the dominant velocity in which the update moves during the optimization process. Sutskever et al. (2013) describe Gradient descent with momentum as

$$v_{i+1} = \mu v_i - \epsilon \nabla \varepsilon(\theta_i) \quad (6)$$

$$\theta_{i+1} = \theta_i + v_{i+1} \quad (7)$$

Where, θ_i is the weight matrix after i updates, v_i is the i^{th} update value, μ is the momentum, ϵ is the step rate of learning and $\nabla \varepsilon(\theta_i)$ is the partial derivative of the error function w.r.t. the parameter θ_i .

Nesterov (1983) introduced Nesterov Accelerated Gradient (NAG) method for effective velocity preservation

in optimization process. In the manner of classical momentum, NAG can be formalized as

$$v_{t+1} = \mu v_t - \epsilon \nabla \varepsilon(\theta_t + \mu v_t) \quad (8)$$

$$\theta_{t+1} = \theta_t + v_{t+1} \quad (9)$$

The small, but key, difference between classical momentum method and NAG is that in the latter, first a partial update to the parameters is done using the last update value, and then the gradient calculation is done for the next update.

The second trick presented by Sutskever et al. (2013) is related to the random initialization of the hidden-to-hidden and input-to-hidden weight matrices. The sparse-ifying technique presented here is inspired by Martens (2010), where all but 15 (or some k) connection weights are set to zero, and the rest are sampled from a Gaussian distribution. The reasoning behind this weight setting has been that a sparse connection matrix would help to diversify the incoming connection from a lower layer.

As a second initialization step, the spectral-radius is kept close to 1, so as to decrease the possibility of the gradients exploding or vanishing over a long time delay, when using sigmoid transfer function.

6.2. Echo-State Networks

It has been argued by Jaeger & Haas (2004) that a random draw from a pre-determined distribution can be used to set the input-to-hidden and hidden-to-hidden connection weights, instead of learning them iteratively. This method, however, is not applied to the hidden-to-output layer connections, which are trained using closed form solutions that involve calculating the pseudo-inverse of a Hessian matrix.

A completely random draw without controlling the distribution parameters might be harmful for setting such weights, though. For instance, if the spectral radius of the hidden-to-hidden weight matrix is much higher or lower than 1, there is a clear possibility that the long term dependency effects are either intractable or vanish, respectively, over time. Hence, we follow the general rule that the spectral radius of the hidden-to-hidden weight matrix is restricted to be close to 1 (1.1, 0.9 etc.) and the input-to-hidden weights are drawn with a small standard deviation of about 0.001.

6.3. Hessian Free Optimization

Martens (2010) propose a second order Hessian-Free (HF) optimization method, inspired by Newton's method, to train deep neural networks with random

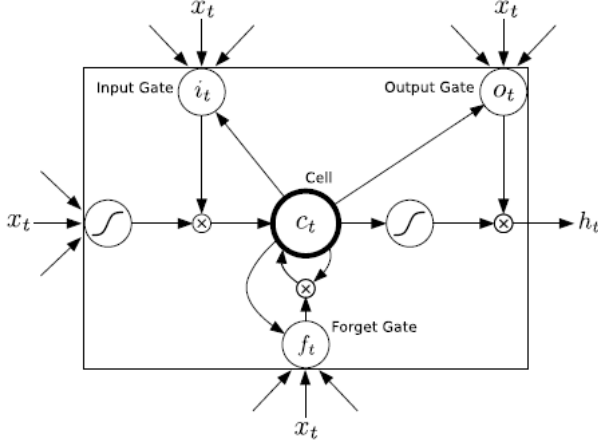


Figure 4. LSTM cell schematic (Graves, 2013)

initializations. HF method obviates the need for pre-training in deep models, which was previously thought to be the most promising way of starting the optimization process, due to the presence of deep pathologies (Hinton et al., 2006; Hinton & Salakhutdinov, 2006).

With respect to the objective function, $f(\theta)$, HF concerns itself with optimizing a simpler sub-objective of $f(\theta)$ by finding local approximations to it. This is done as follows—for a parameter update from θ_n to θ_{n+1} , it optimizes a sub-objective function

$$q_{\theta_n}(\theta) = M_{\theta_n}(\theta) + \lambda R_{\theta_n}(\theta) \quad (10)$$

The term, $M_{\theta_n}(\theta)$, represents a quadratic approximation to $f(\theta_n)$. Normally, $M_{\theta_n}(\theta)$ is chosen to be the Taylor-series expansion of f to second-order terms. This is the same expansion term that is used for Newton’s optimization methods with the key difference that there are no additional assumptions like a low-rank matrix. This would, typically, make the optimization harder since it would involve an inversion of a large matrix. What differentiates HF from other second order optimization methods is that it is made possible to partially optimize $q_{\theta_n}(\theta)$ by conjugate gradient method, instead of gradient descent.

The term $R_{\theta_n}(\theta)$ is a regularization function that penalizes the solution as it moves farther away from θ_n (this modification to the HF method of Martens (2010) was proposed by Sutskever (2013)).

6.4. Long Short-Term Memory (LSTM)

While not particularly a solution to the exploding/vanishing gradients problem, LSTMs (Hochreiter & Schmidhuber, 1997) have been systematically proven (Graves, 2013) to have state-of-the-art perfor-

mance on sequence generation and long-range time series prediction tasks. LSTM alleviates the temporal dependency preservation problem of plain RNNs by structurally modifying the naive neural nodes of the RNN model to produce a more complex LSTM memory cell.

LSTM cell consists of the following novel links, as in Fig. 4, in addition to the conventional hidden units

- *Input gate* to control the in-flow of an input vector into the hidden state. Takes a value from $[0, 1]^N$.
- *Output gate* to control the out-flow of a hidden state activation to the next layer of LSTM-RNN. Takes a value from $[0, 1]^N$.
- *Forget gate* to control the value retention of a memory cell. This link uses the input vector and hidden activation value to determine whether the activation is fed back to the unit for retention over longer time sequences. Takes a value from $[0, 1]^N$.

The original LSTM by Hochreiter & Schmidhuber (1997) uses SGD for training, but it suffers from the exploding gradient problem. In order to solve that, the solution of Graves (2013) uses gradient clipping technique to limit the norm of the gradients and hence stop them from growing too large with time. Even so, the structural complexity of LSTM memory units makes it difficult to implement and harder to train on most systems that do not allow calculation of arbitrary gradients.

6.5. Fast-Dropout RNNs

Wang & Manning (2013) suggest an approximation for dropout (Hinton et al., 2012) in deep neural networks. The suggestion is to treat every neuron as a random variable, whose incoming connections are randomly set to zero, with a probability of $1 - p$. It would be safe to assume that the nature of such a random variable would tend to be Gaussian over sufficiently large number (approximately 10, or more) of incoming connections. The resulting models had orders of magnitude better training times than a naive dropout approach, and the test results matched, and were sometimes better than those of plain MLPs.

Bayer et al. (2013) verified the validity of the fast-dropout approach on RNNs. This was done by concatenating the input-to-hidden and hidden-to-hidden weights into a single array, and applying the same approximation to the incoming connections as in Wang & Manning (2013). Fast-dropout applied to RNNs, works as a regularizer, because the Gaussian approximation of the dropout term leads to a local derivative

of the random variable representation of the node, that acts as an additive regularization term.

The results of Fast-dropout, when applied with the initialization tricks of Sec. 6.1 on standard music datasets, produces state-of-the-art results.

7. Norm-based Regularizers

The first method of regularization in RNN that we evaluate is Tikhonov regularization (Bishop, 1991) on input-to-hidden, recurrent and hidden-to-output weight matrices. It has been claimed in previous RNN related works (Pascanu et al., 2012a) that L1 and L2 penalties on the weight matrix, when added to the cost function of the estimator, may work against improving the long-term dependency remembrance of the network and only partially alleviate the exploding gradients problem.

Using the same example for demonstration as in Sec. 5, we illustrate the effect of L1 and L2 regularizers on the training regime of a time-series network.

8. Stochastic Noise Injection

Noise injection is used as a regularization method in feedforward-only neural networks (Bremermann & Anderson (1991), Flower & Jabri (1993), Jabri & Flower (1992)) to improve generalization. The motive behind adding stochastic noise of different natures to the synaptic weights is to improve fault tolerance in the input and gracefully handle unseen data during prediction.

Adding noise to the weights during optimization works as a regularizer by, essentially, converting the state-space search into a search in a more coarse region of the weight space than what would have been without the additional noise. This property of noisy training has been exploited for training the recurrent weights in RNNs too. By adjusting the weight space to a grainier region, not only are we promised faster convergence but also a cure for the exploding gradients problem. A detailed analysis of Gaussian noise injection in recurrent weight matrix and its behaviour as a regularizer is given in appendix A.

In RNNs, the work of Jim et al. (1996) demonstrate application of stochastic noise to the recurrent layers, much the same way as feedforward MLPs. In the following subsections, we use the additive and multiplicative noise addition model by Jim et al. (1996) to evaluate the performance of a recurrent network in terms of preserving long term dependencies in musical chord sequences. Our analysis of the noisy recurrent

weight training model is followed by noisy input-to-hidden weight model.

8.1. Noise in Recurrent Weights

The first type of noise injection we analyze is in the recurrent weight matrix. In all the analyzed noisy training methods, we restrict ourselves to non-cumulative noise models. In non-cumulative noise methods, the intensity of noise injected at each time-step, t , is independent of the amount of noise injected at $k < t$. As we saw earlier, backpropagation-through-time in RNNs trains essentially the same set of weights in time-space and, hence, we postulate that cumulatively increasing the noise intensity in time space might decrease the convergence performance of the network.

Other than the cumulative nature of the recurrent weight noise, there are two main considerations for deciding the nature of noise that must be injected at each recurrent layer

1. Should the same noise vector be inserted at every time-step in the unrolled representation of the network (per-sequence noise) or a different noise vector be sampled for every time-step (per-time-step noise)?
2. Should the noise be a multiplicative factor of the state of weight vector (multiplicative noise) or simply an additive noise vector sampled from a given distribution (additive noise)?

8.1.1. ADDITIVE NOISE

Additive noise in recurrent weights at time-step, t , is given by

$$W_{hh_t}^* \leftarrow W_{hh_t} + \Delta_{hh_t} \quad (11)$$

$W_{hh_t}^*$ is the modified version of W_{hh_t} after adding the noise term. The noise vector, Δ_{hh} is chosen from a standard normal distribution

$$\Delta_{hh} \sim \mathcal{N}(0, \sigma)$$

In the per-time-step recurrent noise model, we sample a new noise vector, Δ_{hh_t} for every time-step in the unrolled-representation for every iteration of weight update in the optimization process. In the per-sequence recurrent noise model, we sample a new noise vector, Δ_{hh}^* for every iteration in the optimization process and add the same noise to each time-step in the network.

8.1.2. MULTIPLICATIVE NOISE

Multiplicative noise in recurrent weights, analogously, is given by

$$W_{hh_t}^* \leftarrow W_{hh_t} + W_{hh_t} \Delta_{hh_t} \quad (12)$$

The nature of Δ_{hh_t} is the same as before.

As with additive noise, multiplicative noise is also evaluated on the two variants of per-time-step noise and per-sequence noise models.

In both, additive and multiplicative noise models, the perturbation of the weight matrix is done only during the optimization period, and not during forward propagation. During weight training, the original values of the weight matrices are preserved even as noise is added for the gradient calculation for backpropagation-through-time.

8.2. Noise in Feedforward layers

As with noise in the recurrent weight matrix, we would like to close the loop on experimentation by applying the noisy weights training on the feedforward connections too.

During training of feedforward connections with backpropagation of gradients, we use the following weight formulae for noisy weights

$$W_{ih_t}^* \leftarrow W_{ih_t} + \Delta_{ih_t} \quad (13)$$

$$W_{ho_t}^* \leftarrow W_{ho_t} + \Delta_{ho_t} \quad (14)$$

We only work with per-time-step noise model for feedforward layers.

9. Dropout as a Regularizer

Random dropout in MLP connections is used as a generalization technique (Hinton et al., 2012), that works by preventing co-adaptation of multiple features in the training set. A variation of dropout in the activation units is DropConnect (Wan et al., 2013), where random elements from the weight matrix are dropped instead.

We use the DropConnect model on the recurrent weight matrix to try to improve the long-term dependency preserving tendency of our network. As with stochastic noise reduction, dropout in recurrent weights can be applied in two different ways

1. A possibly unique set of weights are dropped out at every time-step (per-time-step dropout).
2. Same set of weights are dropped out at every time-step (per-sequence dropout).

After searching over the range 0–1, we find the best dropout rate suitable for the recurrent connections.

10. Experiments

10.1. Datasets

For evaluating the proposed regularization techniques, we use musical datasets. These are notes based representation of score sheets from four sources—JSB Chorales (harmonized chorales of J.S. Bach), Piano-midi.de (classical music from different sources), Nottingham (folk tunes) and MuseData (classical music).

The dimensionality at each time-step for all four datasets is 88. After dividing the original dataset into training, validation and testing sets (approximately 60%–20%–20% respectively), we split the training and validation samples into chunks of 100 time-steps each. We choose this number because in our experience, for a dataset such as music scores, a length of 100 is long enough to make remembering long term dependencies a necessity while at the same time not making it unreasonably difficult for a network to do so. For samples that are smaller than 100 steps long, we pad them with zeros at the front.

We do no such splitting or prefixing for the test dataset, and use the original sized data chunks for prediction.

10.2. Model Description

Our setup for all four polyphonic music datasets consists of one hidden layer of neurons at each time-step of the RNN. The number of hidden units in the layer is enumerated in the appendix B. The hidden units use the hyperbolic-tangent (tanh) non-linearity and the output nodes use sigmoid. The model parameters are tasked with describing the random variable, \mathbf{y} , such that

$$y_{t,i} = p(x_{t,i} | \mathbf{x}_{1:t-1})$$

Where $x_{t,i}$ denotes the state of note i at time-step t which, if present, is 1 and 0 otherwise.

The loss function which is optimized by this RNN is a mean cross-entropy (CE) loss over all time-steps

$$\begin{aligned} \varepsilon(\theta) = & \frac{1}{T-1} \frac{1}{N} \sum_{i,t,k} x_{t,i}^{(k)} \log y_{t-1,i}^{(k)} \\ & + (1 - x_{t,i}^{(k)}) \log (1 - y_{t-1,i}^{(k)}) \end{aligned}$$

i denotes the note index, t denotes the time-step and k denotes the training sample index.

10.3. Results

On the four datasets, we report the average CE errors in Tab. 1. The results for RNN with norm-based regularizer (RNN-NBR), per-time-step noise (RNN-N), per-sequence noise (RNN-NS), multiplicative noise per-time-step (RNN-MN), multiplicative noise per-sequence (RNN-MNS), dropout per-time-step (RNN-DO), dropout per-sequence (RNN-DOS) and feedforward noise (RNN-FF) are given compared to plain RNNs (with initialization in correct regime) and fast dropout RNN (RNN-FD). Advanced training methods like fast dropout and RNN-NADE (Boulanger-Lewandowski et al., 2012) perform measurably better on this data.

We see that injecting stochastic noise or randomly dropping out weights in recurrent layers during training does not necessarily improve the performance of the RNN training or generalization to the test set. In fact, for most datasets, simply tuning the initialization parameters viz. standard deviation of the weight parameter sampling, sparsification of the weight matrix and spectral radius of the recurrent weight vectors, provides better test performance on the musical datasets, than using the noise injection techniques.

Table 1. Test set results on polyphonic musical datasets

	JSBC	Not.	P-midi	Muse
Plain-RNN	8.58	3.43	7.58	6.99
RNN-FD	8.01	3.09	7.39	6.75
RNN-NBR	8.83	3.70	7.78	8.62
RNN-N	8.92	3.56	7.66	8.40
RNN-NS	8.96	3.58	7.74	8.40
RNN-MN	8.64	3.51	7.71	8.13
RNN-MNS	8.64	3.50	7.70	8.12
RNN-DO	8.48	3.49	7.65	7.98
RNN-DOS	8.55	3.57	7.67	8.00
RNN-FF	8.67	3.54	7.69	8.10

As postulated by Bayer et al. (2013), we observe too that the largest eigenvalue, when training with stochastic noise of dropout in recurrent weights, gets stuck at a lower spectral radius after a fixed number of epochs over multiple tries. There is less incentive for weight matrices with lower spectral radii to change their values by a bigger amount, due to the lack of error information that can be stored over longer time delays. This can be seen in Fig. 5 and Fig. 6. However, this is not the case with norm-based regularizers where the spectral radius continues to grow, albeit very slowly (Fig. 7).

Tab. 2 in appendix B gives the range of values from

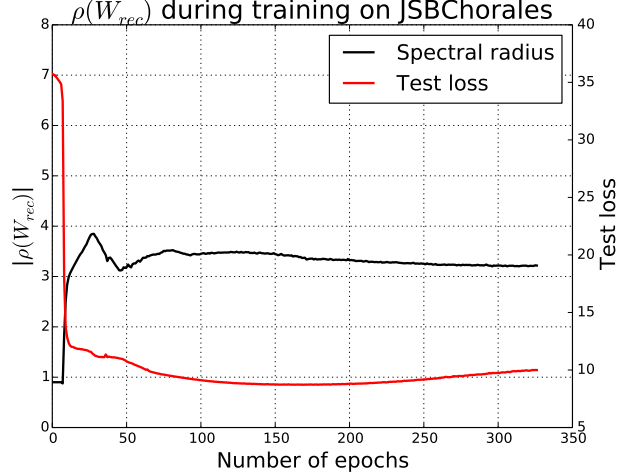


Figure 5. Training with multiplicative noise per-sequence

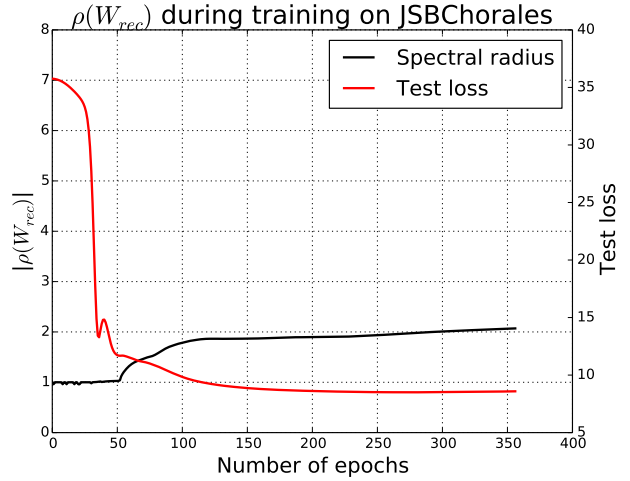


Figure 6. Training with dropout

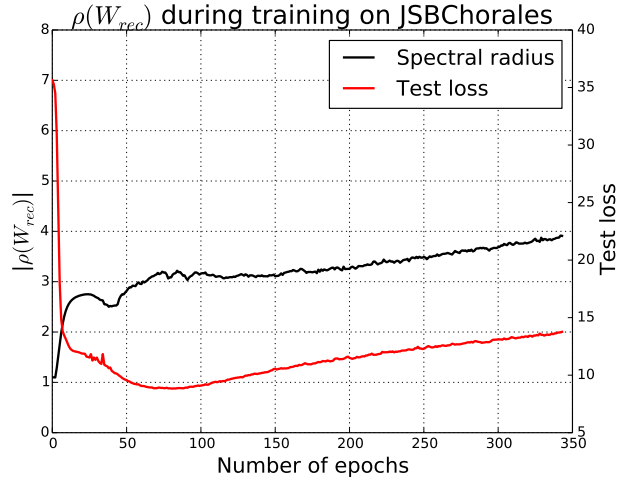


Figure 7. Training with L2 regularizer

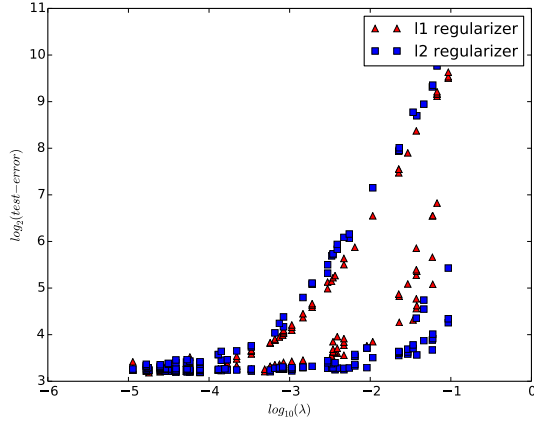


Figure 8. Regularizer λ vs. mean test-error for JSB Chorales

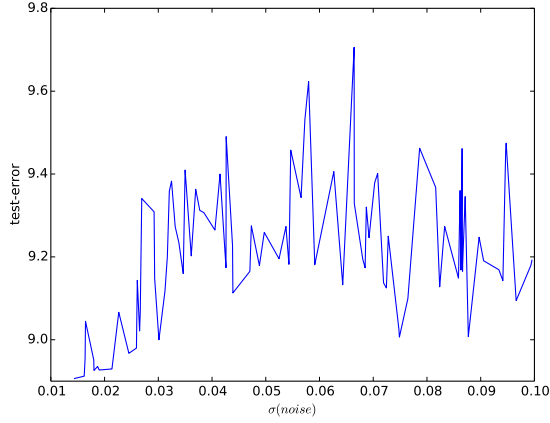


Figure 9. Additive noise σ vs. mean test-error for JSB Chorales

which we generate λ for norm-based regularizer. Fig. 8 shows the average logarithmic test errors over different λ for both, L1 and L2, regularizers. Fig. 9 shows the average test errors over different σ (standard deviation) of additive stochastic noise. The general trend indicates that the network performance decreases as σ increases. Fig. 10 shows the average test errors over different $dropout_p$ (probability that an incoming recurrent weight is set to zero) values, for uniform dropout per-sequence. The general trend indicates that the network performance improves as $dropout_p$ is increased.

11. Conclusion

Through an exhaustive set of experiments with noisy weight updates, random dropout and norm-based reg-

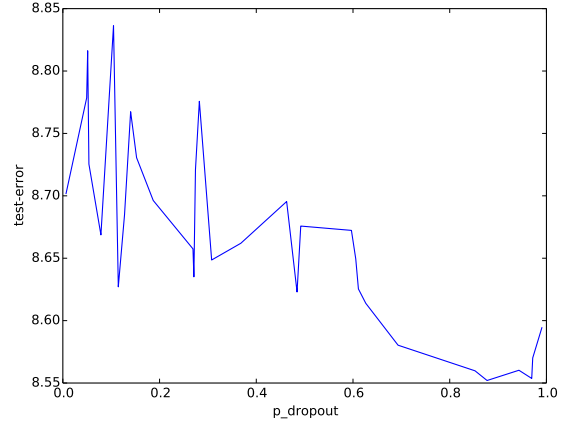


Figure 10. Dropout probability (per-sequence) vs. mean test-error for JSB Chorales

ularization approach we have shown that conjectures about the inefficacy of MLP specific regularizers on RNNs are verifiable. Pascanu et al. (2012a) conjectured that a norm-based penalty on the loss function may reduce the training regime of an RNN to a single point attractor, since the length of the eigenvectors of the weight matrix never exceeded by more than a limited amount. A matrix of weights with such low spectral radius would not suffer from exploding or vanishing gradients at the cost of storing long term dependency effects. We can see this from the demonstration of a simple RNN (Fig. 3). In fact, the analytic presentation of the noisy weight training method shows that noise in weights can also be explained as a loss regularization term.

As the results of stochastic noise, L1 and L2 regularizers on RNNs have not been sufficiently tackled by past works in the field, we believe that we have closed a much needed empirical gap by showing that second order optimization methods, structural solutions or more sophisticated methods of training are indeed imperative to deal with the issues of vanishing gradients and long term dependency in recurrent networks.

References

- Bayer, Justin, Osendorfer, Christian, Chen, Nutan, Urban, Sebastian, and van der Smagt, Patrick. On fast dropout and its applicability to recurrent networks. *arXiv preprint arXiv:1311.0701*, 2013.
- Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

- Bengio, Yoshua, Boulanger-Lewandowski, Nicolas, and Pascanu, Razvan. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 8624–8628. IEEE, 2013.
- Bishop, Chris. Improving the generalization properties of radial basis function neural networks. *Neural Computation*, 3(4):579–588, 1991.
- Boulanger-Lewandowski, Nicolas, Bengio, Yoshua, and Vincent, Pascal. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*, 2012.
- Bremermann, Hans J and Anderson, Russell W. How the brain adjusts synapses maybe. In *Automated Reasoning*, pp. 119–147. Springer, 1991.
- Flower, Barry and Jabri, Marwan. Summed weight neuron perturbation: An $O(n)$ improvement over weight perturbation. In *Advances in Neural Information Processing Systems (NIPS92)*. Citeseer, 1993.
- Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.
- Hinton, Geoffrey, Osindero, Simon, and Teh, Yee-Whye. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.
- Hinton, Geoffrey E and Salakhutdinov, Ruslan R. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
- Hinton, Geoffrey E, Srivastava, Nitish, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan R. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.
- Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- Jabri, Marwan and Flower, Barry. Weight perturbation: An optimal architecture and learning technique for analog vlsi feedforward and recurrent multilayer networks. *Neural Networks, IEEE Transactions on*, 3(1):154–157, 1992.
- Jaeger, Herbert and Haas, Harald. Harnessing non-linearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, 2004.
- Jim, Kam-Chuen, Giles, C Lee, and Horne, Bill G. An analysis of noise in recurrent neural networks: Convergence and generalization. *Neural Networks, IEEE Transactions on*, 7(6):1424–1438, 1996.
- Maas, Andrew L, Le, Quoc V, O’Neil, Tyler M, Vinyals, Oriol, Nguyen, Patrick, and Ng, Andrew Y. Recurrent neural networks for noise reduction in robust asr. In *INTERSPEECH*, 2012.
- Martens, James. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 735–742, 2010.
- Murray, Alan F and Edwards, Peter J. Enhanced mlp performance and fault tolerance resulting from synaptic weight noise during training. *Neural Networks, IEEE Transactions on*, 5(5):792–802, 1994.
- Nesterov, Yurii. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. In *Soviet Mathematics Doklady*, volume 27, pp. 372–376, 1983.
- Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*, 2012a.
- Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. Understanding the exploding gradient problem. *arXiv preprint arXiv:1211.5063*, 2012b.
- Pascanu, Razvan, Gulcehre, Caglar, Cho, Kyunghyun, and Bengio, Yoshua. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013.
- Polyak, Boris Teodorovich. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.
- Rumelhart, David E, Hinton, Geoffrey E, and Williams, Ronald J. Learning representations by back-propagating errors. *Cognitive modeling*, 1988.
- Schaefer, Anton Maximilian, Udluft, Steffen, and Zimmermann, Hans-Georg. Learning long-term dependencies with recurrent neural networks. *Neurocomputing*, 71(13):2481–2488, 2008.
- Sutskever, Ilya. *Training recurrent neural networks*. PhD thesis, University of Toronto, 2013.

Sutskever, Ilya, Martens, James, Dahl, George, and Hinton, Geoffrey. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1139–1147, 2013.

Vincent, Pascal, Larochelle, Hugo, Bengio, Yoshua, and Manzagol, Pierre-Antoine. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pp. 1096–1103. ACM, 2008.

Wan, Li, Zeiler, Matthew, Zhang, Sixin, Cun, Yann L, and Fergus, Rob. Regularization of neural networks using dropconnect. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1058–1066, 2013.

Wang, Sida and Manning, Christopher. Fast dropout training. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 118–126, 2013.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

Appendices

A. Analysis of Noisy Weights

In this section we attempt to show that adding stochastic noise to the weight matrix is equivalent to adding a regularization term to the loss function of the RNN.

Let us define the pre-synaptic activation of the incoming connections to one hidden unit as $a = \mathbf{w}^T \mathbf{x}$. Then, upon adding multiplicative noise to the weight vector, we have

$$a = (\mathbf{w} + \Delta_{\mathbf{m}} \mathbf{w})^T \mathbf{x}$$

$$E[a] = E[(\mathbf{w} + \Delta_{\mathbf{m}} \mathbf{w})^T \mathbf{x}] \quad (15)$$

The noise, $\Delta_{\mathbf{m}}$ is drawn from a zero mean Gaussian ($\Delta_{\mathbf{m}} \sim \mathcal{N}(0, \sigma)$). Additionally, considering \mathbf{w} and \mathbf{x} as constants, we have –

$$E[a] = \mathbf{w}^T \mathbf{x} \quad (16)$$

This shows that the expected value of a is the same as the expected value of pre-synaptic signal that is not perturbed by noise.

For computing the variance of a , we know that,

$$V[AB] = V[A]E[B]^2 + V[B]E[A]^2 + V[A]V[B]$$

Therefore,

$$\begin{aligned} V[a] &= V[(\mathbf{w} + \Delta_{\mathbf{m}} \mathbf{w})^T \mathbf{x}] \\ &= V[\mathbf{w} + \Delta_{\mathbf{m}} \mathbf{w}] \mathbf{x}^2 \\ &\quad + V[\mathbf{x}] E[\mathbf{w} + \Delta_{\mathbf{m}} \mathbf{w}]^2 \\ &\quad + V[\mathbf{w} + \Delta_{\mathbf{m}} \mathbf{w}] V[\mathbf{x}] \end{aligned} \quad (17)$$

The second and third terms on the right hand side of Eq. 17 are zero since the variance in question is that of a constant input, \mathbf{x} .

For the first term of Eq. 17,

$$V[\mathbf{w} + \Delta_{\mathbf{m}} \mathbf{w}] = \mathbf{w}^2 \sigma^2 \quad (18)$$

Where σ is the standard deviation of the Gaussian noise matrix, $\Delta_{\mathbf{m}}$.

Putting this back into Eq. 17, we get

$$V[a] = \sigma^2 (\mathbf{w}^T \mathbf{x})^2 \quad (19)$$

The forms of $E[a]$ and $V[a]$ imply that,

$$\Delta_{\mathbf{m}} \sim \mathcal{N}(0, \sigma) \implies a \sim \mathcal{N}(E[a], V[a]) \quad (20)$$

This means that if the multiplicative noise is assumed to have been sampled from a Gaussian distribution, it is equivalent to assume that the pre-synaptic activations are sampled from a Gaussian.

This equivalence to a sampling form brings us to the sampling form of pre-synaptic activation explained by Bayer et al. (2013), instead of smooth Gaussian approximation.

In place of a , let us use \hat{a} , which we define as –

$$\hat{a} = E[a] + s \sqrt{V[a]}$$

Where, $s \sim \mathcal{N}(0, 1)$.

Using the above incarnation of a to its sampling form, \hat{a} , we may define an effective loss function as follows –

$$\frac{\partial \mathcal{J}}{\partial \hat{a}} \frac{\partial \hat{a}}{\partial w_i} = \frac{\partial \mathcal{J}}{\partial \hat{a}} \left[\frac{\partial \hat{a}}{\partial E[a]} \frac{\partial E[a]}{\partial w_i} + \frac{\partial \hat{a}}{\partial V[a]} \frac{\partial V[a]}{\partial w_i} \right] \quad (21)$$

We will analyse the right hand side of Eq. 21 one at a time.

Consider the first term –

$$\frac{\partial \mathcal{J}}{\partial \hat{a}} \frac{\partial \hat{a}}{\partial E[a]} \frac{\partial E[a]}{\partial w_i} = \frac{\partial \mathcal{J}}{\partial \hat{a}} x_i, \quad (22)$$

Using the expectation value from Eq. 16.

For a pre-synaptic activation, \hat{a} , Eq. 22 is similar to the usual backpropagation term w.r.t a loss function, $\varepsilon^{\hat{a}}$. Therefore, we may simply use the following form of the gradient term –

$$\frac{\partial \varepsilon^{\hat{a}}}{\partial w_i} = \frac{\partial \mathcal{J}}{\partial \hat{a}} x_i \quad (23)$$

Consider now the second term of Eq. 21 –

$$\begin{aligned} \frac{\partial \mathcal{J}}{\partial \hat{a}} \frac{\partial \hat{a}}{\partial V[a]} \frac{\partial V[a]}{\partial w_i} &= \frac{\partial \mathcal{J}}{\partial \hat{a}} \frac{\partial \hat{a}}{\partial \sqrt{V[a]}} \frac{\partial \sqrt{V[a]}}{\partial V[a]} \frac{\partial V[a]}{\partial w_i} \\ &= \frac{\partial \mathcal{J}}{\partial \hat{a}} \cdot s \cdot \frac{1}{2\sqrt{V[a]}} \cdot \frac{\partial (\sigma \mathbf{w}^T \mathbf{x})^2}{\partial (\sigma \mathbf{w}^T \mathbf{x})} \cdot \frac{\partial (\sigma \mathbf{w}^T \mathbf{x})}{\partial w_i} \\ &= s \sigma x_i \frac{\partial \mathcal{J}}{\partial \hat{a}} \end{aligned} \quad (24)$$

This is the same as the post-synaptic gradient term, scaled by the standard deviation of the noise, σ , and independent of the actual weight values.

Hence, we can write Eq. 21 as –

$$\frac{\partial \mathcal{J}}{\partial w_i} = \frac{\partial \varepsilon^{\hat{a}}}{\partial w_i} + \frac{\partial \mathcal{R}_{sampling}^a}{\partial w_i} \quad (25)$$

Where the second term on the right hand side is the regularization term due to multiplicative noise addition to the synaptic weights.

Similar analysis can be done for dropout in recurrent weight matrix, where the Gaussian distribution of the noise vector can be replaced by a Bernoulli distribution approximation when choosing *dropout-p*.

B. Hyper Parameters for RNN Models

For each of the eight RNN models for which the results are listed in Tab. 1 we generate 50 experiments with model hyper parameters chosen from the ranges given in Tab. 2.

The best configurations for all datasets are listed in Tab. 3, Tab. 4, Tab. 5 and Tab. 6.

